GPU-ACCELERATED DISCONTINUOUS GALERKIN METHODS ON POLYTOPIC MESHES*

ZHAONAN DONG†, EMMANUIL H. GEORGOULIS‡, AND THOMAS KAPPAS§

Abstract. Discontinuous Galerkin (dG) methods on meshes consisting of polygonal/polyhedral (henceforth, collectively termed as *polytopic*) elements have received considerable attention in recent years. Due to the physical frame basis functions used typically and the quadrature challenges involved, the matrix-assembly step for these methods is often computationally cumbersome. To address this important practical issue, this work proposes two parallel assembly implementation algorithms on CUDA-enabled graphics cards for the interior penalty dG method on polytopic meshes for various classes of linear PDE problems. We are concerned with both single GPU parallelization, as well as with implementation on distributed GPU nodes. The results included showcase almost linear scalability of the quadrature step with respect to the number of GPU-cores used, since no communication is needed for the assembly step. In turn, this can justify the claim that polytopic dG methods can be implemented extremely efficiently, as any assembly computing time overhead compared to finite elements on 'standard' simplicial or box-type meshes can be effectively circumvented by the proposed algorithms.

Key words. Discontinuous Galerkin, GPU, high order methods, polytopic meshes.

AMS subject classifications. 68Q25, 68R10, 68U05

1. Introduction.

Discontinuous Galerkin (dG) methods have received considerable attention during the last two decades. By combining advantages from both finite element methods (FEMs) and finite volume methods (FVMs) they allow the simple treatment of complicated computational geometries, ease of adaptivity and stability for non-self-adjoint PDE problems [21, 18].

More recently, dG approaches have been shown to be applicable on extremely general computational meshes, consisting of general polytopic elements with an arbitrary number of faces and different local elemental polynomial degrees [14, 7, 12, 10, 13]. A basic feature of these methods is the use of physical frame polynomial bases, as opposed to the standard practice of mapped basis functions in standard finite element implementations. The presence of physical frame basis functions together with the highly involved quadrature requirements over polytopic elements pose new algorithmic complexity challenges in the context of matrix assembly.

The implementation of arbitrary order quadrature rules for non-polynomial integrands over general polytopic domains is highly non-trivial and is addressed in the literature through various techniques. The most general and widely used approach is the subdivision of polytopic elements into basic simplicial or prismatic (with simplicial

* **Funding:** The authors gratefully acknowledge the support by The Leverhulme Trust (grant RPG-2015- 306). Also, this research work was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant" (Project Number: 3270).

†Inria, 2 rue Simone Iff, 75589 Paris, France & CERMICS, Ecole des Ponts, 77455 Marne-la-Vallée 2, France (zhaonan.dong@inria.fr)

‡School of Mathematics and Actuarial Science, University of Leicester, LE1 7RH, United Kingdom (Emmanuil.Georgoulis@le.ac.uk) & Department of Mathematics, School of Applied Mathematical and Physical Sciences, National Technical University of Athens, Zografou 15780, Greece & IACM-FORTH, Heraklion, Crete, Greece

§School of Mathematics and Actuarial Science, University of Leicester, LE1 7RH, United Kingdom (tk223@le.ac.uk)

or hypercubical bases) sub-elements; standard quadrature rules are then employed on each sub-element [14, 7, 10, 13]. Alternative approaches include the use of Euler's formula for homogeneous polynomials, see, e.g., [29, 17, 31, 3], or the direct derivation of quadrature points for general polytopes, see, e.g., [33, 4].

The use of subdivisions, when implemented serially, is typically computationally demanding in this context. At the same time, this approach guarantees the quality of the assembled matrices for nonlinear problems and for problems with localised heterogeneous coefficients. Moreover, in this context quadrature-related variational crimes can be controlled both theoretically and in practice. In contrast, the approaches using Euler's formula [29, 17, 31, 3] are typically faster for quadrature computations on polytopic meshes for *polynomial* integrands only; they are not known to offer safeguarded quadrature error control for *non-polynomial* integrands. As such, they cannot be used with confidence in the assembly of nonlinear problems or for problems with non-polynomial PDE coefficients. Finally, tailored quadrature rules for polytopes [33, 4] typically require costly prior quadrature point optimization steps and, therefore, are not suitable for meshes with highly variable element shapes. Nonetheless, matrix assembly is a *highly parallelizable* process. Thus, it is possible to take advantage of modern computer architectures to achieve highly efficient implementation of all the above approaches.

Graphics Processing Units (GPUs) have been traditionally used for graphics output to a display device. GPUs offer widely available parallel processing capacity, and they are typically more economical in terms of floating point operations per Watt of electricity [22] than CPU clusters of similar parallelization specifications for basic multiply-and-accumulate processes. At the same time, GPUs can only achieve high performance parallelization on substantially more restricted data structures than CPU clusters. Fortunately, the basic multiply-and-accumulate structure of standard quadrature rules can be naturally implemented within the fast operating ranges of modern GPUs; we refer to [5] for a GPU-accelerated implementation for low order conforming elements, or [6] for a respective high order study, showcasing the acceleration potential in this context. Also, in [30] a suite of algorithms and hardware are tested, for low order elements, including usage of atomic operations to avoid race conditions, and mesh 'coloring' allowing invocation of different kernels for differently colored patches.

The benefits of GPU-acceleration in the context of discontinuous Galerkin methods have been studied extensively in the literature over the last decade or so for various classes of electromagnetic, fluid flow and other hyperbolic PDE problems; we refer to [28, 26, 15, 25, 35, 32, 16] for some of the most successful results in the area. The predominant application setting involves explicit time-stepping, e.g., by structure-preserving Runge-Kutta methods, combined with discontinuous Galerkin spatial discretizations with nodal representation of local finite element spaces [20]. Indeed, owing in part to the characteristic minimal communication between elemental spaces, irrespective of the local polynomial degree used, dG methods have shown impressive run-time acceleration when implemented in GPU architectures [28, 15].

The present work is concerned with the development and performance study of GPU-accelerated *assembly algorithms* for dG methods on unstructured meshes comprising *extremely general polytopic elements*. This is achieved via a novel CUDA implementation of the *hp*-version interior penalty dG method for equations with non-negative characteristic form on polytopic meshes in $\mathbb{R}^d$. This class of equations, which includes elliptic, parabolic and first order hyperbolic PDEs, as well as equations of changing type, offers a sufficiently general setting for software development. With regard to shape generality, each element is allowed to be a general polytope with *ar-*

bitrary number of $(d-1)$-dimensional polytopic faces; we refer to [13] for a detailed discussion on the definition and structure of dG methods on polytopic meshes. The element-shape generality requires both new data structures as well as the resolution of new algorithmic challenges, compared to dG implementations on standard simplicial or box-type meshes [28, 26, 15, 25, 35, 32, 16]. The algorithms presented below aim use parallelization within GPU clusters to address the key challenge of reducing the computational cost of arbitrary order quadrature rules over general polytopic domains. Given the extreme scalability potential, the quadratures are performed via subdivisions of the polytopic elements into basic simplicial or prismatic sub-elements. Standard quadrature rules are, in turn, employed on each of these sub-elements before being accumulated into a matrix entry. Correspondingly, for the computation of the face contributions in the present *hp*-version dG setting, subdivision of the $(d-1)$-dimensional faces into simplicial/quadrilateral sub-faces is performed. The choice of quadrature method is made specifically to enable universal applicability: assembly of implicit methods for nonlinear problems or of highly heterogeneous PDE coefficients is possible. We stress, however, that a CUDA implementation of Euler-formula methods, e.g., the one proposed in [3], as by all means possible within the presented algorithmic development. As we shall see below, the excellent scalability of the implementation essentially removes the computational overhead due to subdivision: quadrature kernels require comparable or less time than the sorting algorithms used to process the resulting arrays. To highlight the performance and the versatility of the proposed algorithms, we consider the interior penalty dG method for:

a) fully d-dimensional (dD) problems, $d = 2, 3$, with non-negative characteristic form, approximated on dD unstructured polytopic meshes, and
b) $(dD+1)$ space-time parabolic problems, $d = 2, 3$, approximated on prismatic space-time elements with polytopic bases, with the prism bases perpendicular to the time direction. The dG method in this case is equivalent to a combined dG-timestepping scheme with interior penalty dG discretization in space.

For completeness, we present and compare two distinct algorithmic approaches for the matrix assembly:

1. first compute quadrature values for each simplex in the simplicial subdivision; then combine the values appropriately corresponding to each polytopic element;
2. first precompute the final matrix sparsity patterns, then compute quadrature values and populate the matrices.

The first approach is typical in finite element codes on simplicial or box-type meshes. Interestingly, in the context of general unstructured polytopic meshes with "many" faces per element the first approach produces large number of duplicate values in the sparse array formats; these require further costly sorting and processing. On the other hand, the second approach lends itself more naturally to the case of general polytopic elements with arbitrary number of faces per element. The CUDA implementation of the proposed algorithms is able to achieve small run-times for very large discretizations in both $3D$ and $(2D+1)$ settings. Further, we investigate also the scalability of the second approach in a parallel architecture comprising multiple GPUs; the implementation is carried out using basic Message Passing Interface (MPI) tools. More specifically, the MPI implementation breaks the problem into completely *independent* processes, each assembling for a part of the mesh using a CUDA-enabled GPU. The actual matrix creation takes place through CUDA-enabled GPUs, with each CUDA thread (the smallest execution unit in a CUDA program,) calculating individual matrix entries.

The remainder of this work is structured as follows. The general model PDE problem and some important special cases are presented in Section 2. Section 3 contains detailed description of the approaches 1. and 2. above in the implementation of polytopic dG methods on GPUs, along with some initial numerical experiments highlighting the superior performance of the second approach. Finally, in Section 4, we present a number of challenging numerical experiments on single GPU and multiple GPUs.

2. Model problem and discretization. Let Ω be a bounded open polygonal/polyhedral domain in $\mathbb{R}^d$, $d = 2, 3, 4$.

2.1. Equations with non-negative characteristic form. We consider the advection-diffusion-reaction equation

$$-\nabla \cdot (A\nabla u) + \mathbf{b} \cdot \nabla u + cu = f, \quad \text{in } \Omega, \tag{2.1}$$

where $c \in L_\infty(\Omega)$, $f \in L_2(\Omega)$, and $\mathbf{b} := (b_1, b_2, \ldots, b_d)^\top \in [W^1_\infty(\Omega)]^d$. Here, $A = \{a_{ij}\}_{i,j=1}^d$ is a symmetric positive semidefinite tensor whose entries a_{ij} are bounded, real-valued functions defined on $\bar{\Omega}$, with

$$\boldsymbol{\xi}^\top A(x)\boldsymbol{\xi} \geq 0 \quad \forall \boldsymbol{\xi} \in \mathbb{R}^d, \quad \text{a.e.} \quad x \in \bar{\Omega}.$$

Under the above hypothesis, (2.1) is termed *a partial differential equation with non-negative characteristic form.*

We denote by $\mathbf{n}(x) = \{n_i(x)\}_{i=1}^d$ the unit outward normal vector to $\partial\Omega$ at $x \in \partial\Omega$ and introduce

$$\partial\Omega_0 = \big\{x \in \partial\Omega : \mathbf{n}(x)^\top A(x)\mathbf{n}(x) > 0\big\},$$
$$\partial_-\Omega = \big\{x \in \partial\Omega \backslash \partial\Omega_0 : \mathbf{b}(x) \cdot \mathbf{n}(x) < 0\big\}, \quad \partial_+\Omega = \big\{x \in \partial\Omega \backslash \partial\Omega_0 : \mathbf{b}(x) \cdot \mathbf{n}(x) \geq 0\big\}.$$

The sets $\partial_-\Omega$ and $\partial_+\Omega$ are referred to as the inflow and outflow boundary, respectively. Note that $\partial\Omega = \partial\Omega_0 \cup \partial_-\Omega \cup \partial_+\Omega$. If $\partial\Omega_0$ is nonempty, we subdivide it into two disjoint subsets $\partial\Omega_{\mathrm{D}}$ and $\partial\Omega_{\mathrm{N}}$, with $\partial\Omega_{\mathrm{D}}$ nonempty and relatively open in Ω_0, on which we consider the boundary conditions:

$$u = g_{\mathrm{D}} \quad \text{on} \quad \partial\Omega_{\mathrm{D}} \cup \partial\Omega_-, \quad \mathbf{n} \cdot (A\nabla u) = g_{\mathrm{N}} \quad \text{on} \quad \partial\Omega_{\mathrm{N}}, \tag{2.2}$$

and also adopt the hypothesis that $\mathbf{b} \cdot \mathbf{n} \geq 0$ on $\partial\Omega_{\mathrm{N}}$, whenever $\partial\Omega_{\mathrm{N}}$ is nonempty. Additionally, assuming that there exists a positive constant γ_0 such that $c_0(x)^2 := c(x) - 1/2\nabla \cdot \mathbf{b}(x) \geq \gamma_0$ a.e. $x \in \Omega$, the well-posedness of the boundary value problem (2.1), (2.2) follows.

2.2. Discontinuous Galerkin method. Let $\mathcal{T}_h$ be a subdivision of Ω into disjoint open polygonal or polyhedral elements κ ($d = 2$ or $d = 3, 4$, respectively) such that $\bar{\Omega} = \cup_{\kappa \in \mathcal{T}_h} \bar{\kappa}$ and set $h_\kappa := \mathrm{diam}(\kappa)$. Let also $\mathcal{F}_h$ be the set of all open $(d-1)$–dimensional hyperplanar faces associated with $\mathcal{T}_h$. We write $\mathcal{F}_h = \mathcal{F}_h^I \cup \mathcal{F}_h^B$, with $\mathcal{F}_h^B$ the set of all boundary faces and $\mathcal{F}_h^I$ denotes the set of all interior faces, i.e. the faces shared by two elements. By allowing general polytopic elements in $\mathcal{T}_h$, it is by all means possible that two elements share more than one face. Nonetheless, the term *face* will refer to a $(d-1)$-dimensional planar region of each element, while the term *interface* will refer to the totality of the common boundary between two elements. The domain of all (interior) interfaces will be denoted by $\Gamma_{\mathrm{int}} := \cup_{F \in \mathcal{F}_h^I} F \subset \Omega$.

Given $\kappa \in \mathcal{T}_h$, we write $p_\kappa \in \mathbb{N}$ to denote the *polynomial degree* of the element κ, and collect the p_κ in the vector $\mathbf{p} := (p_\kappa : \kappa \in \mathcal{T}_h)$. We then define the *finite element space* $S^{\mathbf{p}}_{\mathcal{T}_h}$ with respect to $\mathcal{T}_h$ and $\mathbf{p}$ by

$$S^{\mathbf{p}}_{\mathcal{T}_h} := \{u \in L_2(\Omega) : u|_\kappa \in \mathcal{P}_{p_\kappa}(\kappa), \kappa \in \mathcal{T}_h\},$$

where $\mathcal{P}_{p_\kappa}(\kappa)$ denotes the space of polynomials of total degree p_κ on κ. Note that the local elemental polynomial spaces employed within the definition of $S^{\mathbf{p}}_{\mathcal{T}_h}$ are defined in the physical coordinate system, without the need to map from a given reference or canonical frame.

Next, we introduce some trace operators used in the definition of discontinuous Galerkin methods. For element $\kappa \in \mathcal{T}_h$, we define the *inflow* and *outflow* parts of its boundary $\partial\kappa$ by

$$\partial_-\kappa = \{x \in \partial\kappa, \quad \mathbf{b}(x) \cdot \mathbf{n}_\kappa(x) < 0\}, \quad \partial_+\kappa = \{x \in \partial\kappa, \quad \mathbf{b}(x) \cdot \mathbf{n}_\kappa(x) \geq 0\},$$

respectively, with $\mathbf{n}_\kappa(x)$ denoting the unit outward normal vector to $\partial\kappa$ at $x \in \partial\kappa$.

We shall also make use of the *upwind jump* of a function v across a face $F \subset \partial_-\kappa \backslash \partial\Omega$, denoted by

$$\lfloor v \rfloor := v^+_\kappa - v^-_\kappa.$$

Also, for $\kappa_i, \kappa_j \in \mathcal{T}_h$ two adjacent elements sharing a face $F = \partial\kappa_i \cap \partial\kappa_j \subset \mathcal{F}^I_h$, we write $\mathbf{n}_i$ and $\mathbf{n}_j$ to denote the outward unit normal vectors on F, relative to κ_i and κ_j, respectively. Let w be a (scalar- or vector-valued) function and v be a scalar function, that are smooth enough on each element to have a well-defined trace on F from within both $\partial\kappa_i, \partial\kappa_j$. We define the *average* of w and *jump* of v across F by

$$\{\!\{w\}\!\}_F := \frac{1}{2}(w|_{F\cap\partial\kappa_i} + w|_{F\cap\partial\kappa_j}), \quad [\![v]\!]_F := v|_{F\cap\partial\kappa_i} \cdot \mathbf{n}_i + v|_{F\cap\partial\kappa_j} \cdot \mathbf{n}_j,$$

respectively. On a boundary face $F \subset \mathcal{F}^B_h$, with $F \subset \partial\kappa_i$, we simply set $\{\!\{w\}\!\}_F = w|_{F\cap\partial\kappa_i}$, $[\![v]\!]_F := v|_{F\cap\partial\kappa_i} \cdot \mathbf{n}_i$, noting that in the last case $\mathbf{n}_i$ coincides with the unit outward normal vector on the boundary $\partial\Omega$ if the domain is represented exactly by the mesh. We observe that $[\![\cdot]\!]$ and $\lfloor\cdot\rfloor$ may differ only up to a sign. For brevity, we also define the *broken gradient* $\nabla_h v$ of a sufficiently smooth function v to be given by $\nabla_h v|_\kappa = (\nabla v)|_\kappa$ for all $\kappa \in \mathcal{T}_h$.

The symmetric interior penalty discontinuous Galerkin method for (2.1), (2.2) is given by: find $u_h \in S^{\mathbf{p}}_{\mathcal{T}_h}$ such that

$$B(u_h, v_h) = \ell(v_h), \quad \text{for all } \; v_h \in S^{\mathbf{p}}_{\mathcal{T}_h}; \tag{2.3}$$

the bilinear form $B(\cdot,\cdot) : S^{\mathbf{p}}_{\mathcal{T}_h} \times S^{\mathbf{p}}_{\mathcal{T}_h} \to \mathbb{R}$ is given by:

$$\begin{aligned} B(w,v) &:= \int_\Omega \left(A\nabla_h w \cdot \nabla_h v + \mathbf{b} \cdot \nabla_h w v + cwv\right) \mathrm{d}x \\ &- \sum_{\kappa\in\mathcal{T}_h} \int_{\partial_-\kappa\backslash\partial\Omega} (\mathbf{b}\cdot\mathbf{n}) \lfloor w \rfloor v^+ \,\mathrm{d}s - \sum_{\kappa\in\mathcal{T}_h} \int_{\partial_-\kappa\cap(\partial\Omega_{\mathrm{D}}\cup\partial\Omega_-)} (\mathbf{b}\cdot\mathbf{n}) w^+ v^+ \,\mathrm{d}s \\ &- \int_{\Gamma_{\mathrm{int}}\cup\partial\Omega_{\mathrm{D}}} \left(\{\!\{A\nabla w\}\!\} \cdot [\![v]\!] + \{\!\{A\nabla v\}\!\} \cdot [\![w]\!] - \sigma [\![w]\!] \cdot [\![v]\!]\right) \mathrm{d}s, \end{aligned} \tag{2.4}$$

and the linear functional $\ell : S^{\mathbf{p}}_{\mathcal{T}_h} \to \mathbb{R}$ by

$$\begin{aligned}\ell(v) := &\int_\Omega fv \,\mathrm{d}x - \sum_{\kappa\in\mathcal{T}_h} \int_{\partial_-\kappa\cap(\partial\Omega_{\mathrm{D}}\cup\partial\Omega_-)} (\mathbf{b}\cdot\mathbf{n}) g_{\mathrm{D}} v^+ \,\mathrm{d}s \\ &- \int_{\partial\Omega_{\mathrm{D}}} g_{\mathrm{D}} \Big((A\nabla v)\cdot\mathbf{n} - \sigma v \Big) \,\mathrm{d}s + \int_{\partial\Omega_{\mathrm{N}}} g_{\mathrm{N}} v \,\mathrm{d}s.\end{aligned} \tag{2.5}$$

The nonnegative function $\sigma : \Gamma_{\mathrm{int}}\cup\partial\Omega_{\mathrm{D}} \to \mathbb{R}$ appearing in (2.4) and (2.5) is referred to as the *discontinuity-penalization parameter*; its precise definition used in the numerical experiments below is provided in Appendix A.

2.3. An important special case. In this section, we focus on the important subclass of first order (in time) evolution problems which can be considered as a special case of PDEs with non-negative characteristic form. In particular, for $d = 3, 4$, we consider the following special form of $A : \mathbb{R}^d \to \mathbb{R}^{d\times d}$ and $\mathbf{b} : \mathbb{R}^d \to \mathbb{R}^d$,

$$A = \begin{pmatrix} \mathbf{a} & \mathbf{0} \\ \mathbf{0}^\top & 0 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} \mathbf{w} \\ 1 \end{pmatrix},$$

with $\mathbf{a} : \mathbb{R}^s \to \mathbb{R}^{s\times s}$, $s := d-1$, a symmetric non-negative definite tensor and $\mathbf{w} : \mathbb{R}^s \to \mathbb{R}^s$ the spatial wind/advection direction. Substituting this selection into (2.1) gives rise to the classical first order (in time) evolution equations, with the last variable designating the time direction, viz.,

$$\partial_t u - \nabla\cdot(\mathbf{a}\nabla u) + \mathbf{w}\cdot\nabla u + cu = f \quad \text{in } \Omega, \tag{2.6}$$

with $\nabla := (\partial_{x_1}, \ldots, \partial_{x_s})^\top$ and $\nabla\cdot := \sum_{i=1}^s \partial_{x_s}$ the gradient and divergence operators with respect to the spatial variables only, respectively. If $\mathbf{a}$ is additionally uniformly positive definite, i.e.,

$$\boldsymbol{\zeta}^\top \mathbf{a}(t,x)\boldsymbol{\zeta} \geq \theta|\boldsymbol{\zeta}|^2 > 0 \quad \forall\, \boldsymbol{\zeta}\in\mathbb{R}^s, \quad \text{a.e. in } \Omega, \tag{2.7}$$

with θ a positive constant and $x := (x_1, \ldots, x_s)^\top$, (2.6) is, in particular, a parabolic PDE. The preference on temporally implicit high order discretizations for parabolic PDE problems motivates the use space-time dG methods for these problems. Moreover, if spatio-temporal variability is present in the PDE coefficients, matrix assembly has to be performed for every/most time-steps. Hence, it is of interest in the present context to focus also in space-time methods for parabolic problems. To that end, we assume that (2.7) holds for the rest of this section.

Let $\Omega := J \times D$ be a space-time domain with $J := (0, T] \subset \mathbb{R}^+$ a time interval, and $D \subset \mathbb{R}^s$ denoting the spatial domain. Let also ∂D_{D} denote the Dirichlet part of the boundary D. Finally, we set $\partial D_{\mathrm{N}} := \partial D \backslash \partial D_{\mathrm{D}}$ for the Neumann boundary. Thus, we have $\partial\Omega_{\mathrm{D}} = \{(t,x) : t \in J, x \in \partial D_{\mathrm{D}}(x)\}$, and correspondingly for $\partial\Omega_{\mathrm{N}}$.

We consider the linear parabolic problem:

$$\begin{aligned} \partial_t u - \nabla\cdot(\mathbf{a}\nabla u) + \mathbf{w}\cdot\nabla u + cu &= f \quad \text{in } \ \Omega = J\times D, \\ u &= u_0 \quad \text{on } \ \{0\}\times D, \\ u &= g_{\mathrm{D}} \quad \text{on } \ J\times\partial D_{\mathrm{D}}, \\ \mathbf{n}\cdot(\mathbf{a}\nabla u) &= g_{\mathrm{N}} \quad \text{on } \ J\times\partial D_{\mathrm{N}}, \end{aligned} \tag{2.8}$$

with $f \in L_2(J; L_2(D))$, $\mathbf{a} \in [L_\infty(\Omega)]^{s\times s}$, $\mathbf{w} \in [W^1_\infty(\Omega)]^s$, $c \in L_\infty(\Omega)$, $u_0 \in L_2(D)$; the initial condition at $t = 0$, $g_{\mathrm{D}} \in L_2(J; H^{1/2}(\partial D_{\mathrm{D}}))$ and $g_{\mathrm{N}} \in L_2(J; H^{-1/2}(\partial D_{\mathrm{N}}))$;

the Dirichlet and/or Neumann boundary conditions, which may be time-varying, are imposed only on $J \times \partial D$, since the boundary portions $\{0, T\} \times D \subset \partial\Omega \backslash \partial\Omega_0$.

Although it is by all means possible to consider an unstructured space-time mesh for this problem also, we prefer to use the structure of the equation and construct a mesh based on space-time slabs. That way, it is possible to solve for each slab independently of the next ones in the time direction; this idea goes back to [24].

To that end, we begin by introducing a temporal discretization first. Let $\mathcal{I} := \{I_n\}_{n=1}^{N_t}$ be a partition of the time interval J into N_t time steps I_n, with $I_n = (t_{n-1}, t_n]$, for a set of time nodes $\{t_n\}_{n=0}^{N_t}$ with $0 = t_0 < t_1 < \ldots < t_{N_t} = T$. Let also $\tau_n := t_n - t_{n-1}$ denote the length of I_n. Let also $\mathcal{D}_h$ an s-dimensional polytopic mesh subdividing the spatial domain D. Each space-time slab $I_n \times D$, is then subdivided into a mesh $\mathcal{T}_h$ comprising disjoint open prismatic elements $\kappa_n := I_n \times \kappa$, $\kappa \in \mathcal{D}_h$; thus, we have $\mathcal{T}_h := \mathcal{I} \times \mathcal{D}_h$.

For notational simplicity we do not include explicitly 'local' timestepping within one slab in the discussion, i.e., elements $\kappa_n^i := I_n^i \times \kappa$ arising by subdividing I_n into disjoint subintervals I_n^i for some $i = 1, \ldots, m_\kappa$, with m_κ being the maximum number of local stepping on κ_n^i. We stress, however, that this capability is included in the computer implementation presented below. Also, in the present space-time slabbing setting, we assume that the Dirichlet or Neumann *domains* are time-independent; if they are required to be so, we can revert back to the 'monolithic' formulation (2.3). Still, the Dirichlet or Neumann data g_{D} and g_{N}, respectively, are allowed to be time-dependent functions.

Let p_{κ_n} denote the polynomial degree of the space–time element κ_n and let $\boldsymbol{p}_n$ be a vector of the polynomial degrees of all elements in $I_n \times \mathcal{D}_h$. We define the space-time finite element space for the time interval I_n by

$$V^{\boldsymbol{p}_n}(I_n; \mathcal{D}_h) := \{u \in L^2(I_n \times D) : u\big|_{\kappa_n} \in \mathcal{R}_{p_{\kappa_n}}(\kappa_n), \kappa_n \in I_n \times \mathcal{D}_h\},$$

where $\mathcal{R} \in \{\mathcal{P}, \mathcal{PQ}\}$; $\mathcal{P}_{p_{\kappa_n}}(\kappa_n)$ denotes the space of polynomials of space-time total degree p_{κ_n} on κ_n; $\mathcal{PQ}_{p_{\kappa_n}}(\kappa_n)$ is the space of polynomials of degree p_{κ_n} in temporal variable tensor-product with polynomials of total degree p_{κ_n} in all spatial variables. $\mathcal{PQ}_{p_{\kappa_n}}$ is the standard choice when κ are simplices [19], while $\mathcal{P}_{p_{\kappa_n}}$ was proposed in [10] as a reduced complexity choice, without loss of rate of convergence in energy-like norms. The global space-time finite element space $S^{\mathbf{p}}(\mathcal{I}; \mathcal{D}_h)$ is then defined by

$$S^{\mathbf{p}}(\mathcal{I}; \mathcal{D}_h) := \bigoplus_{n=1}^{N_t} V^{\boldsymbol{p}_n}(I_n; \mathcal{D}_h),$$

with $\boldsymbol{p} := (\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n)$ the array of elemental polynomial degrees. Note that the local elemental space-time polynomial spaces are defined in the physical coordinate system, i.e., without mapping from a reference frame; this ensures optimal approximation capabilities over each space-time prism. Finally, let h_{κ_n} denote the diameter of each space-time element κ_n and let $\boldsymbol{h} := (h_{\kappa_n} : \kappa_n \in S^{\mathbf{p}}(\mathcal{I}; \mathcal{D}_h))$ the array of elemental diameters.

The specific space-time slab mesh structure (which is orthogonal with respect to the time direction) allows for reorganization of the face terms in (2.4) and (2.5). In particular, we set $\Gamma_{\mathrm{int}}^n := I_n \times \Gamma_{\mathrm{int}}$, i.e., the union of all element faces in the space-time slab $I_n \times D$ parallel to the time direction. Additional to the operators $\{\!\{\cdot\}\!\}$ $[\![\cdot]\!]$, $\lfloor\cdot\rfloor$, we also specify the *time-jump* $\lfloor u \rfloor_n := u_n^+ - u_n^-$, where $u_n^\pm := \lim_{s \to 0^\pm} u(t_n + s)$, $1 \le n \le N_t$ and $u_0^+ := \lim_{s \to 0^+} u(s)$. We note that the time-jump is just an instance

of the upwind jump $\lfloor\cdot\rfloor$ operator across elemental faces perpendicular to the time direction, i.e., the prismatic element bases. For brevity, we also denote by $(\cdot,\cdot)_\omega$ the $L^2(\omega)$-inner product for any measurable set $\omega\subset\mathbb{R}^d$.

The (space–time) discontinuous Galerkin method for (2.8) then becomes: find $u_h\in S^{\mathbf{p}}(\mathcal{I};\mathcal{D}_h)$ such that:

$$B(u_h,v_h)=\ell(v_h)\quad\text{ for all }\ v_h\in S^{\mathbf{p}}(\mathcal{I};\mathcal{D}_h), \tag{2.9}$$

where

$$B(u,v)=\sum_{n=1}^{N_t}\int_{I_n}\big((\partial_t u,v)_D+B_{\mathrm{S}}(u,v)\big)\,\mathrm{d}t+\sum_{n=2}^{N_t}\big(\lfloor u\rfloor_{n-1},v_{n-1}^+\big)_D+(u_0^+,v_0^+)_D,$$

with the spatial dG bilinear form given by

$$\begin{aligned}B_{\mathrm{S}}(u,v):=&\int_D\big(\mathbf{a}\nabla_h u\cdot\nabla_h v+\mathbf{w}\cdot\nabla_h uv+cuv\big)\,\mathrm{d}x\\&-\sum_{\kappa\in\mathcal{D}_h}\left(\int_{\partial_-\kappa\setminus\partial D}(\mathbf{w}\cdot\mathbf{n})\lfloor u\rfloor v^+\,\mathrm{d}s+\int_{\partial_-\kappa\cap\partial D_{\mathrm{D}}}(\mathbf{w}\cdot\mathbf{n})u^+v^+\,\mathrm{d}s\right)\\&-\int_{\Gamma^n_{\mathrm{int}}\cup\partial D_{\mathrm{D}}}\big(\{\!\{\mathbf{a}\nabla u\}\!\}\cdot[\![v]\!]+\{\!\{\mathbf{a}\nabla v\}\!\}\cdot[\![u]\!]-\sigma[\![u]\!]\cdot[\![v]\!]\big)\,\mathrm{d}s,\end{aligned}$$

and the linear functional ℓ now becomes

$$\ell(v):=\sum_{n=1}^{N_t}\int_{I_n}\left((f,v)_D-\int_{\partial D_{\mathrm{D}}}g_{\mathrm{D}}\big((\mathbf{a}\nabla v)\cdot\boldsymbol{n}-\sigma v\big)\,\mathrm{d}s+\int_{\partial D_{\mathrm{N}}}g_{\mathrm{N}}v\,\mathrm{d}s\right)\mathrm{d}t+(u_0,v_0^+)_D,$$

where $\sigma: J\times(\partial D_{\mathrm{D}}\cup\Gamma_{\mathrm{int}})\to\mathbb{R}_+$ the discontinuity–penalization parameter; see Appendix A for its definition.

The space-time slabs for this problem allows us to solve for each time-step separately. Thus, (2.9) can be solved on each time interval $I_n\in\mathcal{I}$, $n=1,\dots,N_t$. The solution $U=u_h\big|_{I_n}\in V^{\boldsymbol{p}}(I_n;\mathcal{D}_h)$ is given by:

$$\begin{aligned}&\int_{I_n}\big((\partial_t U,W)_D+B_{\mathrm{S}}(U,W)\big)\,\mathrm{d}t+(U_{n-1}^+,W_{n-1}^+)_D\\&=\int_{I_n}\left((f,W)_D-\int_{\partial D_{\mathrm{D}}}g_{\mathrm{D}}\big((\mathbf{a}\nabla W)\cdot\boldsymbol{n}-\sigma W\big)\,\mathrm{d}s+\int_{\partial D_{\mathrm{N}}}g_{\mathrm{N}}W\,\mathrm{d}s\right)\mathrm{d}t+(U_{n-1}^-,W_{n-1}^+)_D,\end{aligned}$$

for all $W\in V^{\boldsymbol{p}}(I_n;\mathcal{D}_h)$, with $U_{n-1}^-:=U(t_{n-1}^-)$ the computed solution from the previous time step, $n=2,\dots,N_T$. For $n=1$, U_0^- is given by the initial condition u_0.

3. Implementation on Graphics Processing Units. Quadrature computations during the assembly of the stiffness and mass matrices are computationally demanding steps, especially for high order local polynomial spaces. Nonetheless, fast methods for stiffness matrix assembly for standard simplicial or box-type element meshes are widely known, especially for low order elements, resulting to (near) optimal scaling. This is *not* the case, however, to the best of our knowledge, for general polytopic element shapes, especially ones arising by aggressive agglomeration of finer simplicial meshes and the use of physically-defined basis functions. Such a development is deemed important to unlock the complexity reduction potential of polytopic

mesh methods, by reducing the cost of assembly, thereby re-instating the linear/non-linear solvers as the dominant source of complexity on par with classical Galerkin approaches. Therefore, fast quadrature evaluations are highly desirable for general polytopic meshes.

The computational cost of assembly becomes extremely relevant when discretizing evolution PDE problems. Then, assembly takes place after each time-step when the PDE coefficients and/or boundary conditions depend on the time variable also. In the latter case, in particular, assembly is typically at least as time-consuming as solving for the approximate solution of each time-step. This is because the resulting algebraic systems arising are either block-diagonal (e.g., for hyperbolic problems or when using explicit schemes for parabolic problems), or the convergence of iterative solvers for implicit computations is typically fast: computer solutions from the previous time-step are excellent starting values typically for the iteration. It is, therefore, highly relevant to develop algorithms able to perform fast assembly for highly agglomerated polytopic element methods.

We will describe two approaches taken to accelerate the quadrature computations for dG methods on general polytopic meshes by the use of Graphics Processing Units (GPUs). As discussed above, our aim is to provide an as general as possible framework, allowing in particular to assemble problems with variable PDE coefficients, stemming from, e.g., non-linear or highly heterogeneous equations.

To that end, the algorithm first subdivides each polytopic element into simplicial sub-elements or, respectively, prismatic ones with simplicial bases in the context of space-time slabs. Correspondingly, for the computation of the face contributions, the algorithm subdivides any co-hyperplanar $(d-1)$-dimensional faces into simplicial sub-faces, or, respectively, rectangular faces for prismatic elements. For brevity, we shall, henceforth, refer collectively as *simplicial subdivision* of an element for both cases of simplicial and prismatic elements with simplicial faces. Standard quadrature rules are then applied on each sub-element or sub-face.

We stress, however, that for the special case of polynomial integrands on polytopic domains, it is possible to further accelerate the quadrature computations within the algorithmic framework presented below, via the use of Euler-formula type quadrature rules [3]. Nonetheless, we prefer to take the point of view of accelerating the computation for polytopic elements in "worst-case" scenarios of practical interest, e.g., heterogeneous coefficients and/or nonlinear problems. Moreover, we are also concerned with developing quadratures for extreme case scenarios of polytopic elements with *many* non-co-hyperplanar polygonal faces each, e.g., elements arising via agglomeration of a finer simplicial subdivision. In these cases, we expect that the majority of face integrals will be performed predominantly over triangular faces, thereby the use of Euler-type quadratures is not expected to be significantly advantageous even in the cases they are applicable.

We note that polytopic elements arising from agglomeration of *many* (e.g., tens or, even, hundreds of) simplicial elements of an underlying finer simplicial mesh can be subdivided in a different fashion to the original constituent simplicial elements for the purposes of quadrature. This, in turn allows to minimize the number of sub-elements on which quadratures will be performed. We now discuss the particulars of the parallel calculation of the element and face integrals on a GPU.

3.1. Computing the integrals. We begin by describing the computation kernels. We implement five kernels: one for volume integrals over each element, and one for each of the face contributions: $\partial\Omega_{\mathrm{D}}$ (or ∂D_{D}), $\partial\Omega_{-}$, $\partial\Omega_{+}$ and Γ_{int} (or $\Gamma^{n}_{\mathrm{int}}$),

respectively. Implementing different kernels for integrals along boundary and interior faces is recommended, as each process requires a different computation load: for instance the kernel computing interior face integrals on Γ_{int} is more computationally demanding since it contributes to four blocks for each interface; we refer to Figure 1 for an illustration.

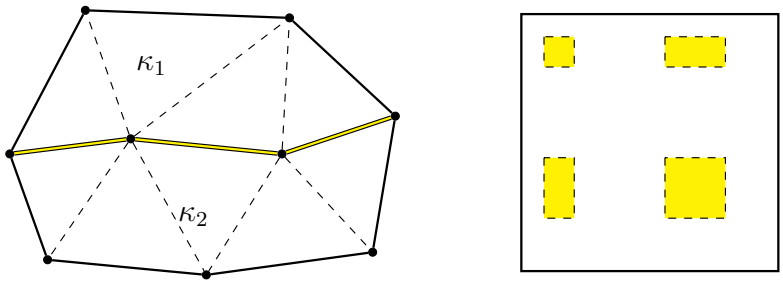

FIG. 1. Left: *Polygonal elements κ_1, κ_2 sharing 3 interior faces.* Right: *Blocks receiving contribution from the shared interior faces.*

We now discuss the structure of the element and interior face integral computations; the boundary integrals are computed in an analogous, yet simpler, fashion. We allocate one thread for the calculation of the volume integral for each simplex of the simplicial subdivision, as illustrated in Figure 2. The implementation allows for locally variable polynomial degree. As such, the algorithm groups simplicial subdivision elements with the same polynomial degree, spawning respective number of threads and invoking the element kernel for each polynomial degree. This allows for the minimisation of idle threads within each block. For the face integral computations, we are presented with the challenge that two elements sharing the same face may admit different degree basis. To address this, we first group the faces according to the maximum of those two degrees and invoke the interior face integral kernels for each maximum degree separately. The use of hierarchical basis functions allows a unified implementation of the kernels for each degree.

In the standard case of simplicial or mapped box-type elements, the use of nodal basis functions is common practice as it offers significant computational savings [20, 28, 26, 15, 25, 35, 32, 16]. This is because nodal basis functions allow for offline precomputation of the evaluation of the basis functions on the quadrature points, which are transferred into the physical domain via elemental maps. For general polytopic elements, nodal basis functions are not an option, at least for elements with many vertices. Instead, we employ a different approach: physical domain polynomial basis functions are defined on a rectangular bounding box of each element before being restricted to the polytope [14, 13]. The basis functions of choice in our implementation are tensorized orthonormal Legendre polynomials of *total* degree p_κ, $\kappa \in \mathcal{T}_h$ (or p_{κ_n}, respectively). As a result, the innermost `for-loop` in Algorithm 3.1 becomes more expensive, as the evaluation (function *evaluate* in Algorithm 3.1) of the basis functions on each quadrature point must be calculated in the physical domain directly.

The *evaluation* function first affinely maps the quadrature points from a reference simplex to each sub-simplex of the simplicial subdivision. Then, using a fast and stable polynomial evaluation method, each polynomial basis function is evaluated at the respective quadrature points. The polynomial evaluation implementation benefits from the fused multiply–add (FMA) operation capability of the GPU [23, 9]. Moreover, storing polynomial coefficients and reference simplex quadrature points in

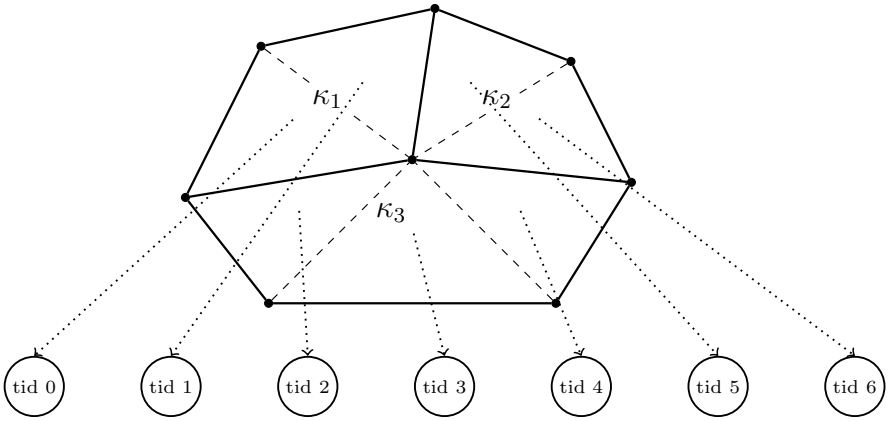

FIG. 2. *Three polygonal elements split into 7 simplices. One thread is used per simplex and correspondingly for the face integrals.* `tid` *refers to "thread index."*

Algorithm 3.1 Pseudocode for the calculation of the bilinear form for one simplex.

```
read geometry data
for i = 1 to #Trial basis functions do
    for j = 1 to #Test basis functions do
        val_ij ← 0
        for k = 1 to #Quadrature points do
            u_i = evaluate(i, q_k)
            v_j = evaluate(j, q_k)
            val_ij ← val_ij + B(u_i, v_j)
        end for
        write val_ij into global memory
    end for
end for
```

the GPU's *constant memory* can improve memory throughput, as threads belonging to the same warp (a group of 32 threads with contiguous indices) are all executing the polynomial evaluation for the same data set in each loop. Constant memory size of current GPU chips (typically 64KB at the time of writing) is sufficient to hold all the data needed for implementation of local polynomial degree up to $p = 10$ in three dimensions.

3.2. Connectivity. Upon deciding on a simplicial subdivision for the polytopic mesh, we discuss different potential choices for index and quadrature value allocation manipulations. We consider two different *approaches* for the assembly of the stiffness and mass matrices in terms of index manipulation:

1. first compute the volume and face quadrature values for each simplex in the simplicial subdivision; then combine the values appropriately corresponding to each polytopic element, or
2. first precompute the sparsity pattern of the final stiffness and mass matrices, then compute the volume and face quadrature values for each simplex in the simplicial subdivision and, subsequently, populate the matrices with the calculated values.

As we shall see below, each of the above general approaches offer advantages on

different settings.

3.3. Approach 1. The general idea of the first approach is to calculate the quadrature values for each simplex of the simplicial subdivision, along with their row and column indices in an unsorted coordinate format containing, typically, duplicates. Subsequently, we convert the three arrays into sparse format, ready for porting into a linear solver.

In particular, we create three arrays to store row index, column index and quadrature value. Each thread uses its own memory space on these arrays to store the computed integrals. Careful assignment of the correct memory space on each thread can drastically improve the memory store performance, taking advantage of the coalesced memory accesses. Thus, it is possible to achieve 100% memory store efficiency. This is illustrated in Figure 3, where we depict the memory access pattern of each thread. Arrows with the same type (solid, dashed, or dotted lines, respectively) belong to the same iteration. Therefore, if they belong to the same warp, they write simultaneously. As we can see, threads with contiguous indices write in a coalesced fashion, which typically accelerates memory operations.

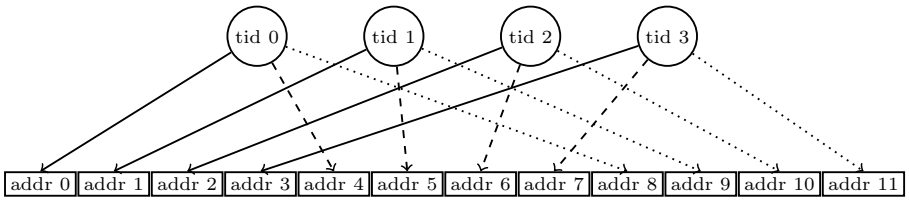

FIG. 3. *Memory storage pattern of Approach 1.*

A key advantage of this first approach is that write operations can be arranged in a coalesced manner, thereby increasing performance. Moreover, since each simplex of the simplicial subdivision is regarded as a stand-alone element, mesh partitioning when using multiple GPUs (or processors, in general,) becomes immediate. At the same time, the presence of duplicates results into fewer elements and faces being calculated per kernel invocation. Most importantly, however, creating the final sparse matrix structure out of the three arrays becomes increasingly expensive, with the increase in the number of duplicates.

We investigate the practical performance of Approach 1, by computing the complete assembly of one time-step of the space-time stiffness matrix arising when using the dG method (2.9) on a linear parabolic problem. For each polynomial degree p the mesh was chosen such that the resulting matrix could fit in the global memory of a single GPU card. On average 3 triangles are agglomerated to construct a polytopic element. All numerical investigations use a single Tesla P100 PCIe card with 16GB of global memory, having a total of 3584 cores, with a total peak double precision performance of 4.67 teraFLOPS. The host machine has a pair of 14-core Intel Xeon E5-2680v4 with 256GB of DDR4 memory.

In Table 1 we collect the assembly times recorded using single and double precision arithmetic, respectively. In the first three rows of each case the *execution time per million degrees of freedom (DoFs)* for the elements, the interior faces, and the inflow faces, respectively, is presented. The total execution time for all the kernels is given, which also includes the (negligible) execution time for the boundary faces and the imposition of Dirichlet conditions. For the CUDA kernel timings in this work, we used the `nvprof` tool and we reported the average of five invocations for each kernel.

	$p=1$	$p=2$	$p=3$	$p=4$	$p=5$
single precision					
element kernel	0.007	0.059	0.27	1	3
interior kernel	0.003	0.024	0.087	0.29	0.7
inflow kernel	0.002	0.014	0.054	0.16	0.4
total kernels	0.014	0.1	0.41	1.4	4.1
indices	4.3	8.6	18.5	31.8	49
total assembly	4.7	9.6	20.6	36	57
double precision					
element kernel	0.008	0.067	0.31	1.2	3.7
interior kernel	0.005	0.039	0.14	0.45	1.1
inflow kernel	0.003	0.016	0.06	0.18	0.5
total kernels	0.017	0.12	0.51	1.8	5.3
indices	4.7	10	18.5	30	55.2
total assembly	5.2	11.3	21.3	35	66.4

TABLE 1
Approach 1: seconds per million degrees of freedom *using single and double precision, respectively.*

The rows "indices" record the time needed to build the sparse matrix out of the three arrays (row, column, value), which includes sorting of the indices as well as addition and removal of the duplicate quadrature values. For the index sorting step, we employ the `csr_matrix` class from Python's SciPy [34] sparse module upon transferring all data back to the host memory from the GPU memory. Note that this built-in sorting method is single-threaded and dominates the total time. We prefer to include the cost of the index sorting step in the discussion, even though it is not a primal concern of the proposed work; its performance will highlight a number of important observations. Of course, it may be possible to reduce the runtime by the use of multi-threaded sorting, or investigate other options. Nevertheless, we prefer not to pursue this direction further at this point, since our primary interest lies in the assembly of polytopic meshes consisting of "many" simplices per element. As such, the bottle-neck of manipulation of duplicates poses a related severe challenge, which will be addressed below.

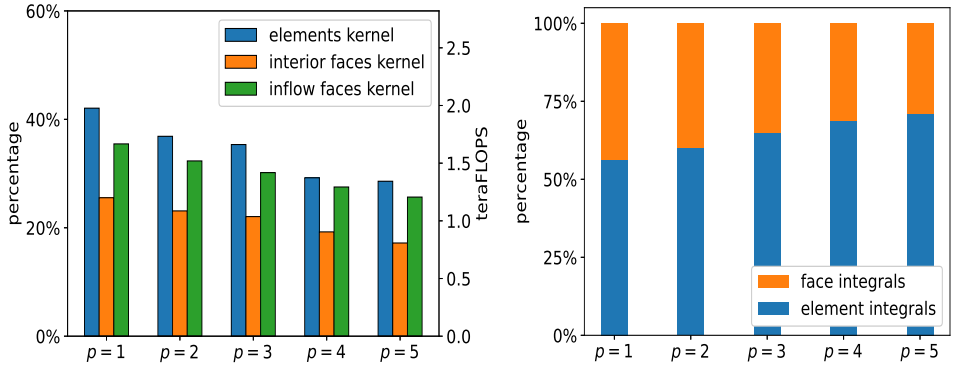

FIG. 4. *Approach 1: (left) double precision performance percentage and FLOPS; (right) percentage of computation time for element vs. face kernels, for* $p = 1, \ldots, 5$.

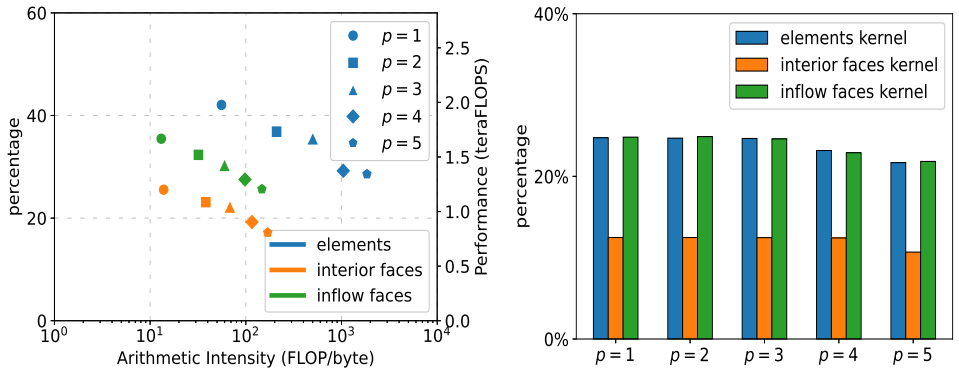

FIG. 5. *Approach 1: (left) performance in teraFLOPS over arithmetic intensity for transactions to/from DRAM; (right) achieved occupancy for the three main kernels.*

In any case, the implementation is able to assemble a million degrees of freedom using linear basis functions in less than 5 seconds, with the integral evaluation taking only 14 milliseconds. As the polynomial degree p grows, the assembly times grow, due to the higher cost per degree of freedom by the decreasing sparsity and the increasing quadrature cost of the resulting system matrix. The "total assembly" is the time required to provide the complete system matrix in compressed sparse row (CSR) format, ready for the linear solver.

In Figure 4 (left panel), we record the performance achieved by each kernel. The maximum performance of more than 40%, translating into 1.9 teraFLOPS on the Tesla P100 is achieved by the element integral kernel for $p = 1$. In Figure 4 (right panel) we present a comparison of the total execution time for element versus the combined face integrals. The total execution time is increasingly dominated by the element integral kernel as the order of approximation increases: each dD-simplicial sub-element quadrature requires $O(p^d)$ operations, while simplicial face integrals need $O(p^{d-1})$; cf. Figures 4 and 7 below.

We note, however that, due to the number of duplicate values, this approach may not be recommended for meshes comprised of polytopic elements with *many* faces per element, e.g., ones arising from agglomeration procedures; see Figure 9 below for an example of such mesh.

3.4. Approach 2. In contrast, following the second approach, we compute first the non–zero indices and delete the duplicates, before storing them into a sparse format for fast access. Subsequently, we calculate the quadrature values for each simplex of the simplicial subdivision and we atomically store them into their target positions. The key idea of the second approach is to first precompute the sparsity pattern of the stiffness and mass matrices and, subsequently, write the calculated quadrature values directly into their final position. By doing so, we can calculate more elements in the same kernel invocation, since no duplicate values are stored in the GPU's global memory. Moreover, in this way we make use of the special structure to accelerate the creation of the sparsity pattern. This is particularly relevant for polytopic meshes with *many* faces per element as we shall see below.

The matrix of the resulting linear system from the dG discretization has a natural sparse block-structure. Given that the number of basis functions depends *only* on the polynomial degree (and *not* on the particular element shape) the index set for

each element and face is precomputed through the knowledge of the local polynomial degree. Thus, we can avoid creating duplicate (global) indices for the elements of the simplicial subdivision. Crucially, the same principle applies to the face integral computations, whereby *all* the face integrals of the common interface (containing many faces per element) between two elements are stored on the same blocks. For example, in a mesh of approximately $500k$ triangles agglomerated into $8k$ elements with $p = 3$, following Approach 1 of creating all the indices for each individual interior face and converting them into CSR format took $6.4s$. In contrast, on the same processor it took $1.7s$, using Approach 2, whereby we allocate a unique index per common element interface (containing many faces).

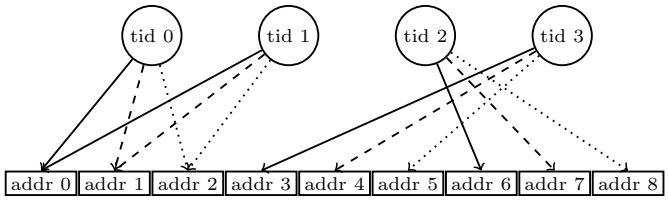

FIG. 6. *Memory storage pattern of Approach 2 with scattered memory operations. Lines with same type are write operations that are performed simultaneously. Threads 0 and 1 are calculating integrals of the same polygonal element, hence they write in the same memory locations with atomic operations.*

	$p=1$	$p=2$	$p=3$	$p=4$	$p=5$
single precision					
element kernel	0.007	0.061	0.27	1	3
interior kernel	0.019	0.09	0.2	0.36	0.8
inflow kernel	0.002	0.015	0.06	0.16	0.4
total kernels	0.029	0.17	0.52	1.5	4.2
indices	0.63	1.2	2.3	3.9	7
total assembly	0.79	1.5	3.1	6	12.2
double precision					
element kernel	0.008	0.069	0.3	1.2	3.7
interior kernel	0.018	0.083	0.2	0.53	1.2
inflow kernel	0.003	0.017	0.07	0.19	0.5
total kernels	0.03	0.17	0.57	1.9	5.5
indices	0.84	1.5	2.7	4.4	7.7
total assembly	1.02	1.9	3.7	7.2	14.7

TABLE 2
Approach 2: seconds per million degrees of freedom using single and double precision, respectively.

Upon creation, the matrix is transferred to the device memory for population with the calculated quadrature values. To identify the position where each value must be added, a binary search is required. Care must be taken when adding the contributions of every simplex or face to rule out, so-called, race conditions. Since it is possible that simplices belonging to the same polygon can also belong to the same execution warp, they will try to update the same memory locations in global memory simultaneously, as is illustrated in Figure 6. To avoid this, these writes must

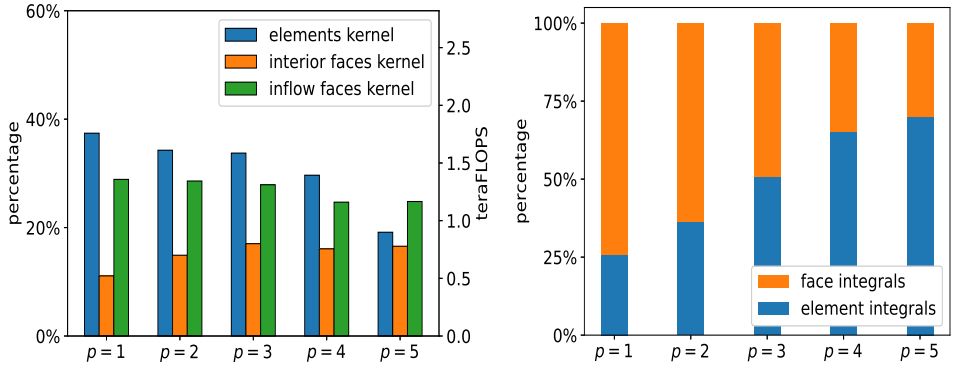

FIG. 7. *Approach 2: (left) double precision performance percentage and FLOPS; (right) percentage of computation time for element vs. face kernels, for* $p = 1, \ldots, 5$.

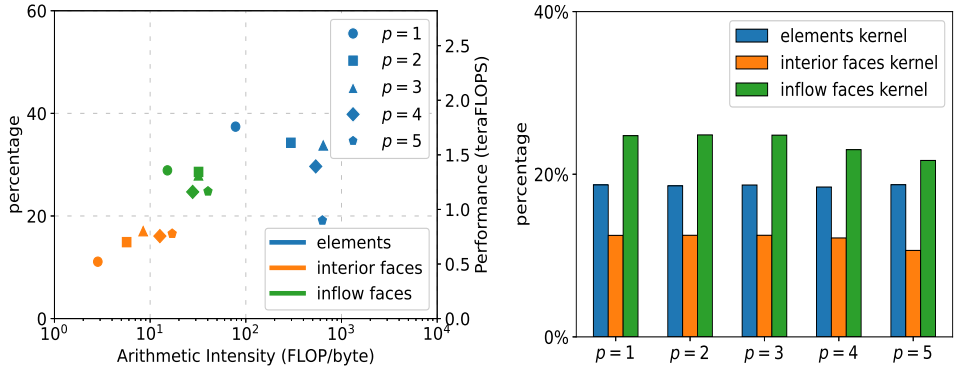

FIG. 8. *Approach 2: (left) performance in teraFLOPS over arithmetic intensity for transactions to/from DRAM; (right) achieved occupancy for the three main kernels.*

be made as atomic operations. It is worth mentioning here that since floating point addition is not associative and there is no a priori guarantee on the order of the atomic additions, different matrices (up to floating point precision) may be created after multiple executions of the same exact problem. Nonetheless, in the numerous numerical investigations we performed, this issue appears to have negligible effect on the results.

In Table 2 we collect the assembly times recorded using single and double precision arithmetic, respectively. Here, the "indices" row records the time spent on creating, sorting and converting the indices into the CSR format, which now precedes the execution of the kernels. With this approach, the time to assemble a million degrees of freedom, using linear basis functions is less than 1 second, almost six times faster than Approach 1. In Figure 7 (left panel), we can see the performance achieved by each kernel. The scattered memory access pattern affects negatively the performance, mainly for low order methods, compared to Approach 1. For high order methods calculating the quadrature value becomes more expensive compared to the memory operations to store it in its final position, so the scattered memory access pattern has a marginal effect. Finally, in Figure 7 (right panel), a comparison of the total execution time for element versus face integrals is recorded. The face integral kernel dominates

the total execution time for low order elements, resulting to a modest increase in the total kernel execution time, compared to Approach 1. Nonetheless, the significant savings recorded in the index manipulation of Approach 2, (cf. Tables 1 and 2,) showcases its viability. We note that both approaches use the same index sorting functions of Python's SciPy module. More sophisticated/parallel implementation of index sorting routines has the potential to improve the "indices" (and, of course, "total") times for both approaches. Indeed, as we shall see in Section 4.2 below, the index sorting run time can be reduced via an implementation on multiple GPUs.

	Approach 1	Approach 2
elements	126	144
interior faces	198	210
inflow faces	109	107

TABLE 3
Number of 32bit registers used per thread, as reported by `nvprof`.

Remark 3.1. The arithmetic intensity results in Figures 5 and 8 (left panels) suggest that we are, in most cases, in the compute bound region. The theoretical occupancy for all kernels, based on the selected threads per block and used registers per thread, matches the achieved occupancy, cf. right panels in Figures 5 and 8. While it would be possible to reduce the number of registers per thread in an effort to further increase performance, such a direction would limit considerably the sought-after generality of the kernels, which is central to this work: the kernels have been designed to handle essentially arbitrary polynomial degrees and element shapes.

4. Numerical experiments. We continue the investigation of the performance of the algorithms by testing them on two computational problems. The first deals with the question of performance on meshes comprising elements with *many* faces each, constructed by aggressive agglomeration of a fine background mesh. The second series of numerical experiments investigates the performance and scalability of an implementation of the algorithms on multiple GPUs.

4.1. Performance on highly agglomerated meshes. Highly agglomerated meshes, i.e., meshes arising from agglomerating *many* simplices are relevant in many applications [13], such as domains with highly heterogeneous boundaries [11, 7] and also in the context of multilevel solvers [1, 2]. Approach 1 is not suitable in this setting due to the excessive number of duplicate entries. Therefore, we seek to test whether Approach 2 has a performance penalty when the mesh consists of highly agglomerated elements, due to the high number of atomic operation replays arising by large numbers of threads updating the same memory locations simultaneously. Nevertheless, the benefit from not creating duplicate indices of Approach 2 in this context is expected to offer superior overall performance of the assembly process.

We start with a problem defined on the domain Ω with oscillating boundary, which is approximating $(0,1)^2$. To represent the computational domain, we employed $503,596$ unstructured simplicial elements as the background mesh; we refer to Figure 9 for an illustration of the extreme agglomeration process resulting to a 30-element polygonal mesh. We note that exactly the same dG method on exactly these meshes have been used in [11] for the numerical approximation of a convection-diffusion problem, where optimal convergence rates have been observed and recorded. In our computations below, the original simplicial mesh is agglomerated into $8,337$ and $125,981$

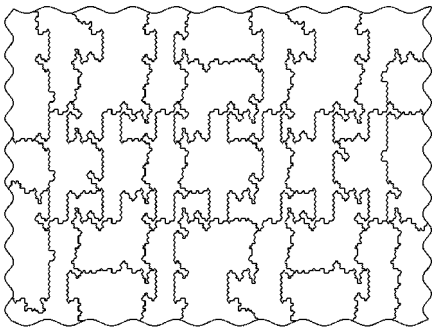

FIG. 9. 503,596 *triangles agglomerated into* 30 *polygonal elements.*

elements respectively. We also test the same method by assembling on the original mesh of 503,596 simplices, treating each simplex as an element. The characteristics of the meshes can be found in Table 4. The resulting $2D$ polygonal meshes are tensorised into forming $3D$ prismatic space-time elements. On the latter, we assemble the space-time dG method (2.9) with the reduced-complexity $\mathcal{P}_{p_{\kappa_n}}$ Galerkin space choice, for the numerical approximation of the linear parabolic problem (2.8) with the specifics: $\mathbf{a} = I_2$, $\mathbf{w} = (0,0)^\top$, $c = 1$. The load function f is selected so that the solution $u(x,y,t) = \sin(\pi x)\sin(\pi y)(1-t)$.

	Mesh 1	Mesh 2	Mesh 3
#elements	8,337	125,981	503,596
#triangles	503,596	503,596	503,596
#interior faces	124,628	376,455	754,118

TABLE 4

The original triangular mesh with 503,596 *triangles, and meshes constructed via two levels of agglomeration of the original mesh.*

For this test we used a single Tesla P100 PCIe card with 16GB of 4096 bit HBM2 global memory, with a total of 3584 cores and processing power of 4.67 teraFLOPS. In Figure 10, we record the performance achieved by Approach 2 in double precision. Similar performance is observed in all cases. The element integrals kernel achieved between 35% and 40% of peak performance with a maximum of about 1.85 teraFLOPS. The interior faces kernel performance was between 12% to 16% of the peak with a maximum computational throughput of 750 gigaFLOPS. We also observe a performance penalty when using the original $504k$-element mesh in the inflow faces kernel. This is due to the scattered memory reads of the solution from the previous time-step. This overhead does not appear for the two meshes with 8,337 and 125,981 agglomerated polygonal prisms. This is due to contiguous threads that calculate quadratures from simplices belonging to the same element, reading the same solution coefficients from the previous time-step.

In Table 5 we record the *kernel* and *total* assembly times for $p = 1,2,3$ using double precision. Although it is not necessary to use the very fine background mesh for the quadrature evaluation (it is enough to use any sub-triangulation into as many simplices as faces), we do so in this numerical experiment to highlight further the

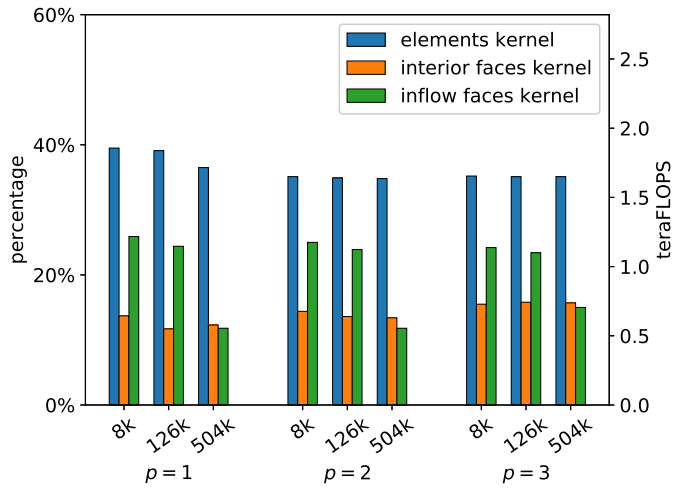

FIG. 10. *Double precision performance of the three main kernels on the* 3 *meshes from Table 4, for* $p = 1, 2, 3$.

acceleration potential of the proposed approach. This results to the execution time for the element kernels to be the same in all cases. As expected, differences in performance arise from the interior faces kernel, as the highly agglomerated meshes have far fewer interior faces than the background mesh. Of course, more aggressive agglomeration results to coarser meshes and therefore fewer global degrees of freedom. We envisage to apply such meshes within a mesh-adaptive Galerkin framework, thereby equilibrating the local resolution requirements for a given accuracy with the computational cost.

4.2. Performance on multiple GPUs. To assess the performance and scalability of the proposed algorithms on larger scale problems, we consider a 3D problem with non-negative characteristic form (2.1) with coefficients $A = 0.01I_3$, $\mathbf{b} = (1+x, 1+y, 1+z)^\top$, $c = 3+xyz$, with $\Omega = (0,1)^3$. The load function f is selected so that the solution $u(x,y,z) = \sin(\pi x)\sin(\pi y)\sin(\pi z)$. Note that different choices of A and $\mathbf{b}$ result to different sets of active face terms on each element; the above choice is a typical scenario. The polyhedral elements stem from the agglomeration of a fine three-dimensional unstructured tetrahedral mesh. This is in contrast to the prismatic meshes used for the parabolic problem above, as this is now a fully $3D$ unstructured grid. As such, we expect both more kernel evaluations (more terms in the bilinear form) and higher connectivity (more non-zero entries).

To estimate the scalability of the assembly process, we implement Approach 2 on clusters consisting of multiple GPUs per node. A Message Passing Interface (MPI) implementation distributes the load to each GPU card and each GPU is responsible for computing a part of the global matrix. Specifically, the following processes are implemented:

1. METIS [27] is used to subdivide the mesh in $Ngpus$ parts. We assign to each polyhedron a weight to minimize the communication cost and to simultaneously balance the quadrature cost among the GPUs;
2. we flag the interior faces on the boundaries of the partition created by METIS. Those faces are processed with a modified interior faces kernel;
3. each GPU creates the sparsity pattern of its allocated subdivision. Instead of

	$p=1$	$p=2$	$p=3$
Mesh 1: $8k$ elements			
total degrees of freedom	$33k$	$83k$	$167k$
element kernel	0.006	0.12	1
interior kernel	0.003	0.04	0.3
inflow kernel	0.002	0.03	0.2
total kernels	0.012	0.2	1.5
indices	0.48	0.7	1.3
total assembly	1.34	1.7	3.7
Mesh 2: $126k$ elements			
total degrees of freedom	$504k$	$1.3m$	$2.5m$
element kernel	0.006	0.12	1
interior kernel	0.012	0.14	0.9
inflow kernel	0.002	0.03	0.2
total kernels	0.02	0.3	2.1
indices	1.15	3.2	10.7
total assembly	1.9	4.6	15
Mesh 3: $504k$ elements			
total degrees of freedom	$2m$	$5m$	$10m$
element kernel	0.006	0.12	1
interior kernel	0.022	0.3	1.8
inflow kernel	0.005	0.05	0.3
total kernels	0.033	0.5	3.1
indices	2	7.4	25.7
total assembly	2.9	9.6	32.9

TABLE 5

Approach 2: seconds using double precision arithmetic for the highly agglomerated meshes and for one time-step.

SciPy's built-in CPU routines, (as done in the single GPU examples above,) we use the CUDA Thrust library [8] to perform the index manipulation steps directly on the GPUs;

4. assembly of the partial stiffness matrices takes place on each GPU by executing the quadrature evaluation kernels.

The MPI implementation of Approach 2 has two significant benefits. First, each GPU creates only a partial matrix in CSR format, allowing for much larger problems to be assembled in the same runtime. Also this allows to build in parallel stiffness matrices that are too large to fit in the global memory of one single card. Moreover, as index sorting is performed separately for each (smaller) partial matrix, the index computation cost is reduced. The partial matrices can then be used for local matrix–vector product operations on each GPU before communicating partial solutions with cards holding neighbouring subdivisions. This is particularly pertinent in the context of multilevel/domain decomposition algorithms.

The quadrature times recorded here include the kernels' execution time (as per table in the previous section) *and* also the memory transfers from RAM to the global memory of GPUs. This is done in an effort to showcase realistic assembly times. The results showcase reduction in both index manipulation and quadrature evaluation as

#GPUs	1	2	4	8	16
$p = 1$, $1.57m$ elements ($6.28m$ DoFs), $6.29m$ tetrahedra					
index manipulation	7.4	4.5	2.7	1.6	1.3
quadrature evaluation	2	1.7	1.7	1.5	1.4
total assembly	9.4	6.2	4.4	3.1	2.7
$p = 2$, $1.57m$ elements ($15.7m$ DoFs), $6.29m$ tetrahedra					
index manipulation	28.5	14.5	8.3	4.5	2.7
quadrature evaluation	9.7	5	3.9	2.5	2.3
total assembly	38.2	19.5	12.2	7	5
$p = 3$, $197k$ elements ($3.94m$ DoFs), $786k$ tetrahedra					
index manipulation	15.1	7.7	4.7	2.7	1.6
quadrature evaluation	5.7	3.5	2.6	2.1	1.9
total assembly	20.8	11.2	7.3	4.8	3.5

TABLE 6
Time in seconds for the assembly of the fully 3D problem using Approach 2 and double precision.

the number of GPUs is increased. Timings for the MPI code were performed using Python's `perf_counter()` from the `time` module, upon calling an `MPI_Barrier()`. Therefore, each reported time is the maximum across all MPI processes. An interesting observation is that index manipulation run times using standard, freely available algorithms can be balanced with quadrature kernel execution times in multiple GPU architectures.

Acknowledgements. We gratefully acknowledge the availability of the ALICE High Performance Computing Facility at the University of Leicester and also resources by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 EPSRC funding (capital grant EP/P020259/1), and DiRAC STFC funding (www.dirac.ac.uk).

Appendix A. The discontinuity-penalization parameter. For completeness, we now give a precise formula for the discontinuity-penalization function σ appearing in the IP-dG formulations (2.3) and (2.9). The stability and error analysis of the IP-dG method under this choice of penalization, as well as a detailed discussion on the practical relevant of this choice, can be found in [13, 11].

Obviously, each polytopic element $\kappa \in \mathcal{T}_h$ (or $\kappa \in \mathcal{D}_h$, respectively,) can be covered by different families of simplices $\mathcal{K}_\kappa := \{K_j\}_{j=1}^{m_\kappa}$, of possibly different cardinalities, i.e., we have $\kappa \subset \cup_{K_j \in \mathcal{K}_\kappa} K_j$. Each such family $\mathcal{K}_\kappa$ will be referred to as a *covering* of κ and we shall denote by $\mathbb{K}_\kappa$ the set of all such coverings. For instance, any subtriangulation of κ is a valid covering; equally coverings with overlapping simplices are also valid. Let h_ω, ρ_ω and $|\omega|$ denote the diameter, the inscribed radius, and the sD-volume of a domain $\omega \subset \mathbb{R}^s$, $s = 1, \ldots, d$, respectively. We say that an element κ is *p-coverable* if there exists at least one covering $\mathcal{K}_\kappa$ of κ, such that: 1) $h_{K_j} \sim h_\kappa$ $\rho_{K_j} \sim \rho_\kappa$, *and* 2) $\max_{\mathbf{x}\in\partial\kappa, \mathbf{z}\in\partial K_j} |\mathbf{x} - \mathbf{z}| \leq \rho_{K_j}/(8p)^2$, for all $j = 1, \ldots, m_\kappa$, with $|\cdot|$ denoting the Euclidean distance. In other words, an element is p-coverable if there exists a covering $\mathcal{K}_\kappa$ comprising simplices each with similar shape-regularity to the original element, which cover κ within a distance at most $\rho_{K_j}/(8p)^2$ each, away from the element's boundary. In [11], a considerably weaker concept of p-coverability is used allowing, in particular, K_j to be general *curved* prisms.

Now let $F \in \mathcal{F}_h^I$ a face shared by two elements $\kappa_1, \kappa_2 \in \mathcal{T}_h$; if $F \subset \partial\Omega_{\mathrm{D}}$, we set $\kappa_2 = \emptyset$. Denote also by $K^F \in \underline{\mathbb{K}}_\kappa$ a sub-simplex having F as face also. We define the discontinuity-penalization parameter on F by

$$\sigma|_F := C_\sigma \max_{\kappa\in\{\kappa_1,\kappa_2\}} \left\{ \min\left\{ \frac{|\kappa|}{\sup_{K^F\in\underline{\mathbb{K}}_\kappa} |K^F|}, C_{\mathrm{cov}}(\kappa) \right\} \frac{\bar{a}_\kappa p_\kappa^2 |F|}{|\kappa|} \right\},$$

for a computable constant $C_\sigma > 0$, with $\bar{a}_\kappa := \|\mathbf{n}^\top \mathbf{A}\mathbf{n}\|_{L_\infty(\kappa)}$, (correspondingly $\bar{a}_\kappa := \|\mathbf{n}^\top \mathbf{a}\mathbf{n}\|_{L_\infty(\kappa)}$), and $C_{\mathrm{cov}}(\kappa) := p_\kappa^{2(d-1)}$ if κ is p-coverable, or $C_{\mathrm{cov}}(\kappa) := \infty$ if not.

REFERENCES

[1] P. F. Antonietti, A. Cangiani, J. Collis, Z. Dong, E. H. Georgoulis, S. Giani, and P. Houston, *Review of discontinuous Galerkin finite element methods for partial differential equations on complicated domains*, in Building bridges: connections and challenges in modern approaches to numerical partial differential equations, vol. 114 of Lect. Notes Comput. Sci. Eng., Springer, [Cham], 2016, pp. 279–308.

[2] P. F. Antonietti, P. Houston, X. Hu, M. Sarti, and M. Verani, *Multigrid algorithms for hp-version interior penalty discontinuous Galerkin methods on polygonal and polyhedral meshes*, Calcolo, 54 (2017), pp. 1169–1198.

[3] P. F. Antonietti, P. Houston, and G. Pennesi, *Fast numerical integration on polytopic meshes with applications to discontinuous Galerkin finite element methods*, J. Sci. Comput., 77 (2018), pp. 1339–1370.

[4] E. Artioli, A. Sommariva, and M. Vianello, *Algebraic cubature on polygonal elements with a circular edge*, Comput. Math. Appl., 79 (2020), pp. 2057–2066.

[5] K. Banaś, F. Krużel, and J. Bielański, *Finite element numerical integration for first order approximations on multi- and many-core architectures*, Comput. Methods Appl. Mech. Engrg., 305 (2016), pp. 827–848.

[6] K. Banaś, P. Płaszewski, and P. Macioł, *Numerical integration on GPUs for higher order finite elements*, Comput. Math. Appl., 67 (2014), pp. 1319–1344.

[7] F. Bassi, L. Botti, A. Colombo, D. A. Di Pietro, and P. Tesini, *On the flexibility of agglomeration based physical space discontinuous Galerkin discretizations*, J. Comput. Phys., 231 (2012), pp. 45–65.

[8] N. Bell and J. Hoberock, *Chapter 26 - thrust: A productivity-oriented library for cuda*, in GPU Computing Gems Jade Edition, W. mei W. Hwu, ed., Applications of GPU Computing Series, Morgan Kaufmann, Boston, 2012, pp. 359 – 371.

[9] S. Boldo and J. Muller, *Some functions computable with a fused-mac*, in 17th IEEE Symposium on Computer Arithmetic (ARITH-17 2005), 27-29 June 2005, Cape Cod, MA, USA, 2005, pp. 52–58.

[10] A. Cangiani, Z. Dong, and E. H. Georgoulis, *hp-version space-time discontinuous galerkin methods for parabolic problems on prismatic meshes*, SIAM J. Sci. Comput., 39 (2017), pp. A1251–A1279.

[11] A. Cangiani, Z. Dong, and E. H. Georgoulis, *hp-version discontinuous galerkin methods on essentially arbitrarily-shaped elements*, Submitted for publication. arXiv:1906.01715, (2019).

[12] A. Cangiani, Z. Dong, E. H. Georgoulis, and P. Houston, *hp-version discontinuous Galerkin methods for advection-diffusion-reaction problems on polytopic meshes*, ESAIM Math. Model. Numer. Anal., pp. 699–725.

[13] A. Cangiani, Z. Dong, E. H. Georgoulis, and P. Houston, *hp-version discontinuous Galerkin methods on Polygonal and Polyhedral Meshes*, Springer, 2017.

[14] A. Cangiani, E. H. Georgoulis, and P. Houston, *hp-version discontinuous Galerkin methods on polygonal and polyhedral meshes*, Math. Models Methods Appl. Sci., 24 (2014), pp. 2009–2041.

[15] J. Chan, Z. Wang, A. Modave, J.-F. Remacle, and T. Warburton, *GPU-accelerated discontinuous Galerkin methods on hybrid meshes*, J. Comput. Phys., 318 (2016), pp. 142–168.

[16] J. Chan and T. Warburton, *GPU-accelerated Bernstein-Bézier discontinuous Galerkin methods for wave problems*, SIAM J. Sci. Comput., 39 (2017), pp. A628–A654.

[17] E. B. Chin, J. B. Lasserre, and N. Sukumar, *Numerical integration of homogeneous functions on convex and nonconvex polygons and polyhedra*, Comput. Mech., 56 (2015), pp. 967–981.

[18] B. Cockburn, G. E. Karniadakis, and C. Shu, *The development of discontinuous Galerkin methods*, in Discontinuous Galerkin methods (Newport, RI, 1999), vol. 11 of Lect. Notes Comput. Sci. Eng., Springer, Berlin, 2000, pp. 3–50.

[19] K. Eriksson, C. Johnson, and V. Thomée, *Time discretization of parabolic problems by the discontinuous Galerkin method*, RAIRO Modél. Math. Anal. Numér., 19 (1985), pp. 611–643.

[20] J. S. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods*, vol. 54 of Texts in Applied Mathematics, Springer, New York, 2008. Algorithms, analysis, and applications.

[21] P. Houston, C. Schwab, and E. Süli, *Discontinuous hp-finite element methods for advection-diffusion-reaction problems*, SIAM J. Numer. Anal., pp. 2133–2163.

[22] S. Huang, S. Xiao, and W. Feng, *On the energy efficiency of graphics processing units for scientific computing*, 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, pp. 1–8.

[23] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*, Aug. 2008.

[24] P. Jamet, *Galerkin-type approximations which are discontinuous in time for parabolic equations in a variable domain*, SIAM J. Numer. Anal., 15 (1978), pp. 912–928.

[25] A. Karakus, N. Chalmers, K. Świrydowicz, and T. Warburton, *A GPU accelerated discontinuous Galerkin incompressible flow solver*, J. Comput. Phys., 390 (2019), pp. 380–404.

[26] A. Karakus, T. Warburton, M. H. Aksel, and C. Sert, *A GPU-accelerated adaptive discontinuous Galerkin method for level set equation*, Int. J. Comput. Fluid Dyn., 30 (2016), pp. 56–68.

[27] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[28] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven, *Nodal discontinuous Galerkin methods on graphics processors*, J. Comput. Phys., 228 (2009), pp. 7863–7882.

[29] J. B. Lasserre, *Integration on a convex polytope*, Proc. Amer. Math. Soc., 126 (1998), pp. 2433–2441.

[30] G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin, *Finite element assembly strategies on multi-core and many-core architectures*, Internat. J. Numer. Methods Fluids, 71 (2013), pp. 80–97.

[31] S. E. Mousavi and N. Sukumar, *Numerical integration of polynomials and discontinuous functions on irregular convex polygons and polyhedrons*, Comput. Mech., 47 (2011), pp. 535–554.

[32] J.-F. Remacle, R. Gandham, and T. Warburton, *GPU accelerated spectral finite elements on all-hex meshes*, J. Comput. Phys., 324 (2016), pp. 246–257.

[33] A. Sommariva and M. Vianello, *Product Gauss cubature over polygons based on Green's integration formula*, BIT, 47 (2007), pp. 441–453.

[34] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*, Nature Methods, 17 (2020), pp. 261–272.

[35] Y. Xia, J. Lou, H. Luo, J. Edwards, and F. Mueller, *OpenACC acceleration of an unstructured CFD solver based on a reconstructed discontinuous Galerkin method for compressible flows*, Internat. J. Numer. Methods Fluids, 78 (2015), pp. 123–139.